Simulated Annealing

A Montecarlo Algorithm for Non-convex Optimization

Gianluca Covini

University of Pavia

January 22, 2024

Indice

General setting

The problem we want to solve is a minimization problem:

Optimization Problem

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned}$$

where $f : \mathbb{R}^p \longrightarrow \mathbb{R}$ and $\mathcal{X} \subseteq \mathbb{R}^p$.

General setting

The problem we want to solve is a minimization problem:

Optimization Problem

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned}$$

where $f : \mathbb{R}^p \longrightarrow \mathbb{R}$ and $\mathcal{X} \subseteq \mathbb{R}^p$.

In general, we need to approximate the solution whenever the function f is complex and the global minimum is difficult to find analitically.

Non-convex problems

Many algorithms struggle when the function has more than one local minimum, i.e. when the function is not convex.

1D Ackley's function

$$f : \mathbb{R} \longrightarrow \mathbb{R}$$

$$f(x) = -ae^{-b\sqrt{x^2}} - e^{\cos(cx)} + a + e$$

where we chose as parameters $a = 15$, $b = 0.1$ and $c = 2\pi$.

Non-convex problems

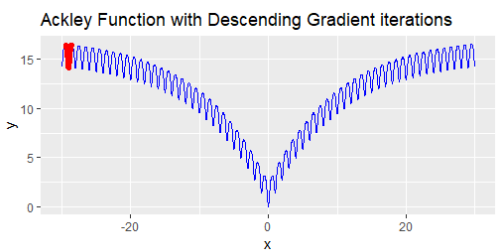
Many algorithms struggle when the function has more than one local minimum, i.e. when the function is not convex.

1D Ackley's function

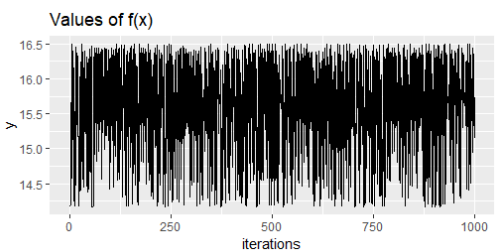
$$f : \mathbb{R} \longrightarrow \mathbb{R}$$

$$f(x) = -ae^{-b\sqrt{x^2}} - e^{\cos(cx)} + a + e$$

where we chose as parameters $a = 15$, $b = 0.1$ and $c = 2\pi$.



(a) Iterations GD



(b) Values plot GD

Non-convex problems

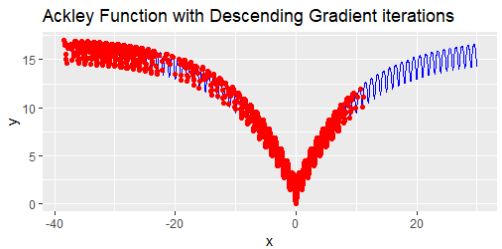
Many algorithms struggle when the function has more than one local minimum, i.e. when the function is not convex.

1D Ackley's function

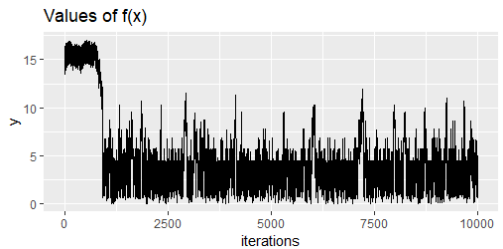
$$f : \mathbb{R} \longrightarrow \mathbb{R}$$

$$f(x) = -ae^{-b\sqrt{x^2}} - e^{\cos(cx)} + a + e$$

where we chose as parameters $a = 15$, $b = 0.1$ and $c = 2\pi$.



(a) Iterations GD - Long Step



(b) Values plot GD - Long Step

Simulated Annealing

Simulated Annealing Algorithm uses randomness to overcome the problem: the algorithm takes inspiration from the process of annealing of materials.

- f: internal energy of the system
- x: system configuration.

Simulated Annealing

Simulated Annealing Algorithm uses randomness to overcome the problem: the algorithm takes inspiration from the process of annealing of materials.

- f: internal energy of the system
- x: system configuration.

- The algorithm then simulates the process of **heating** of the material: in this way the particles move from initial equilibrium state (that could be a local minimum);

Simulated Annealing

Simulated Annealing Algorithm uses randomness to overcome the problem: the algorithm takes inspiration from the process of annealing of materials.

- f: internal energy of the system
- x: system configuration.

- The algorithm then simulates the process of **heating** of the material: in this way the particles move from initial equilibrium state (that could be a local minimum);
- The following slow **cooling** process let, then, the particles to set in a lower energy state (that should be the global minimum of the energy function).

Algorithm

1: We choose an initial point $x = x_{in}$ and an initial high value of temperature $t = t_{in}$;

Algorithm

1: We choose an initial point $x = x_{in}$ and an initial high value of temperature $t = t_{in}$;
2: We generate randomly a new point x';

Algorithm

1: We choose an initial point $x = x_{in}$ and an initial high value of temperature $t = t_{in}$;
2: We generate randomly a new point x';
3: If the value of $y' = f(x')$ is lower than the previous value $y = f(x)$ we accept the new point $(x = x')$, otherwise we accept the new point with probability $e^{-(y'-y)/t}$;

Algorithm

1: We choose an initial point $x = x_{in}$ and an initial high value of temperature $t = t_{in}$;
2: We generate randomly a new point x';
3: If the value of $y' = f(x')$ is lower than the previous value $y = f(x)$ we accept the new point $(x = x')$, otherwise we accept the new point with probability $e^{-(y'-y)/t}$;
4: We save the best solution so far;

Algorithm

1: We choose an initial point $x = x_{in}$ and an initial high value of temperature $t = t_{in}$;
2: We generate randomly a new point x';
3: If the value of $y' = f(x')$ is lower than the previous value $y = f(x)$ we accept the new point $(x = x')$, otherwise we accept the new point with probability $e^{-(y'-y)/t}$;
4: We save the best solution so far;
5: Steps $2 - 4$ are repeated for a fixed number of iterations (or until an equilibrium is reached);

Algorithm

1: We choose an initial point $x = x_{in}$ and an initial high value of temperature $t = t_{in}$;
2: We generate randomly a new point x';
3: If the value of $y' = f(x')$ is lower than the previous value $y = f(x)$ we accept the new point $(x = x')$, otherwise we accept the new point with probability $e^{-(y'-y)/t}$;
4: We save the best solution so far;
5: Steps $2 - 4$ are repeated for a fixed number of iterations (or until an equilibrium is reached);
6: Temperature t is decreased following an annealing schedule;

Algorithm

1: We choose an initial point $x = x_{in}$ and an initial high value of temperature $t = t_{in}$;
2: We generate randomly a new point x';
3: If the value of $y' = f(x')$ is lower than the previous value $y = f(x)$ we accept the new point $(x = x')$, otherwise we accept the new point with probability $e^{-(y'-y)/t}$;
4: We save the best solution so far;
5: Steps $2 - 4$ are repeated for a fixed number of iterations (or until an equilibrium is reached);
6: Temperature t is decreased following an annealing schedule;
7: Steps $2 - 6$ are repeated until $f(x)$ no longer sensibly decrease.

```
        while (i < maxit & abs(delta_y) > eps) {
            x_[i] = x[i] + rnorm(1, mean, sd)
            y_[i] = f(x_[i])

            delta_y = y_[i] - y[i]

            if (delta_y <= 0 | runif(1) < exp(-delta_y/t[i])){
              x[i+1] = x_[i]
              y[i+1] = y_[i]
            }
            else {
              x[i+1] = x[i]
              y[i+1] = y[i]
            }

            if (y_[i] < y_best[i]){
              x_best[i+1] = x_[i]
              y_best[i+1] = y_[i]
            }
            else {
              x_best[i+1] = x_best[i]
              y_best[i+1] = y_best[i]
            }

            if (i %% N == 0){
              t[i+1] = a_s(t_in, t[i], i)
            }
            else{
              t[i+1] = t[i]
            }
            i = i+1
        }

```

Listing: Simulated Annealing Loop

Input

Our `simulated_annealing` routine takes as input:

Input

- `f`: the 1D Ackley's function shown at the beggining;
- `a_s`: the annealing schedule (when not specified we take logarithmic annealing);
- `x_in`: the initial guess x_{in} (we took `x_in = -29`);
- `mean`, `sd`: the parameters of the gaussian that generates the new samples (if not specified `mean = 0` and `sd = 1`);
- `t_in`: the initial temperature (if not specified `t_in = 1`);
- `maxit`: the maximum number of iterations, set to 10^5;
- `eps`: the solution tolerance, set to 10^{-5};
- `N`: the number of iterations before cooling, set to 10.

For the sake of reproducibility we set a seed of 2111.

Annealing schedule

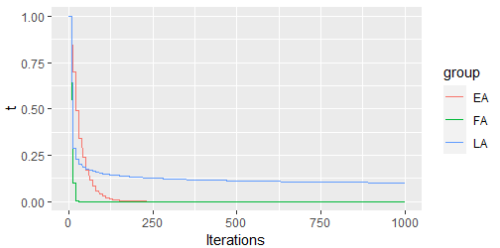
Annealing schedules

$$t(k+1) = \frac{\ln(2)}{\ln(k+1)} t_{in} \quad t(k+1) = \gamma t(k) \quad t(k+1) = \frac{1}{k+1}$$

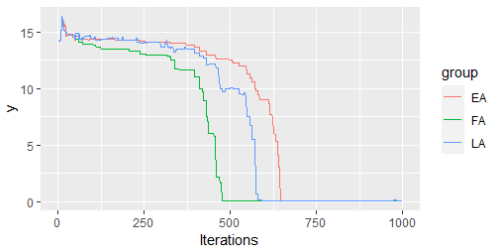
Annealing schedule

Annealing schedules

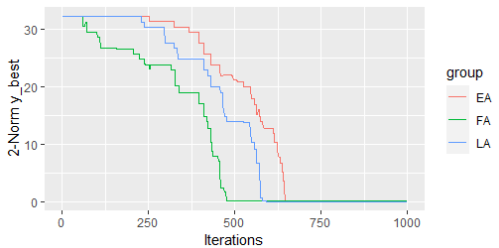
$$t(k+1) = \frac{\ln(2)}{\ln(k+1)} t_{in} \quad t(k+1) = \gamma t(k) \quad t(k+1) = \frac{1}{k+1}$$



(a) Temperature over iterations



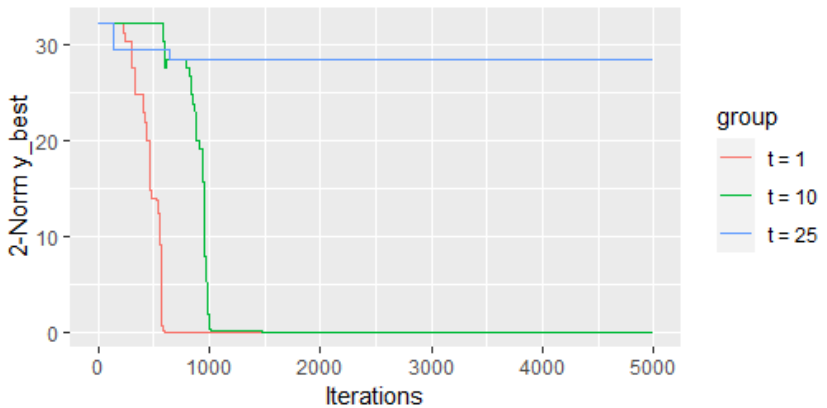
(b) Value of f over iterations



(c) Error on iterations

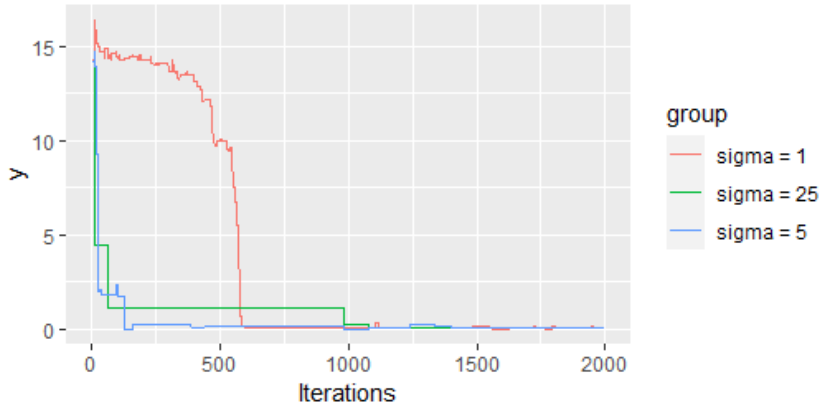
Initial conditions

(a) Value of f over iterations

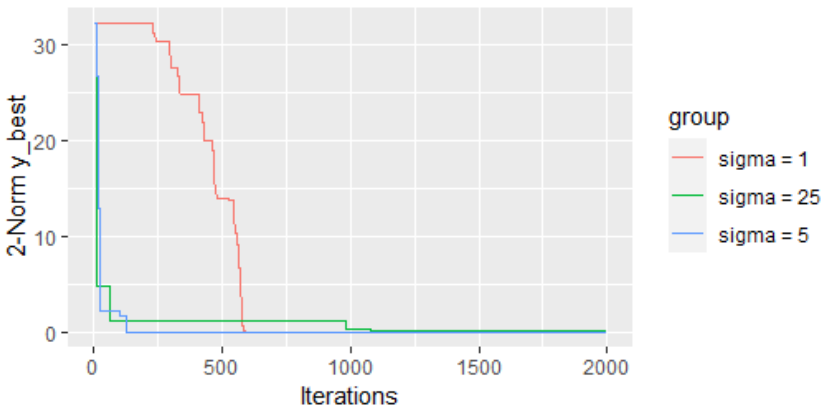


(b) Error on iterations

Random parameters



(a) Value of f over iterations



(b) Error on iterations